Linux Kernel Programming
System Calls

Pierre Olivier

Systems Software Research Group @ Virginia Tech

March 23, 2017

Virginia Tech

Outline

1. System calls: general notions
2. Syscall invocation: user space side
3. Syscall execution: kernel space side
4. Implementing a new system call

Virginia Tech

Outline

1 System calls: general notions

2 Syscall invocation: user space side

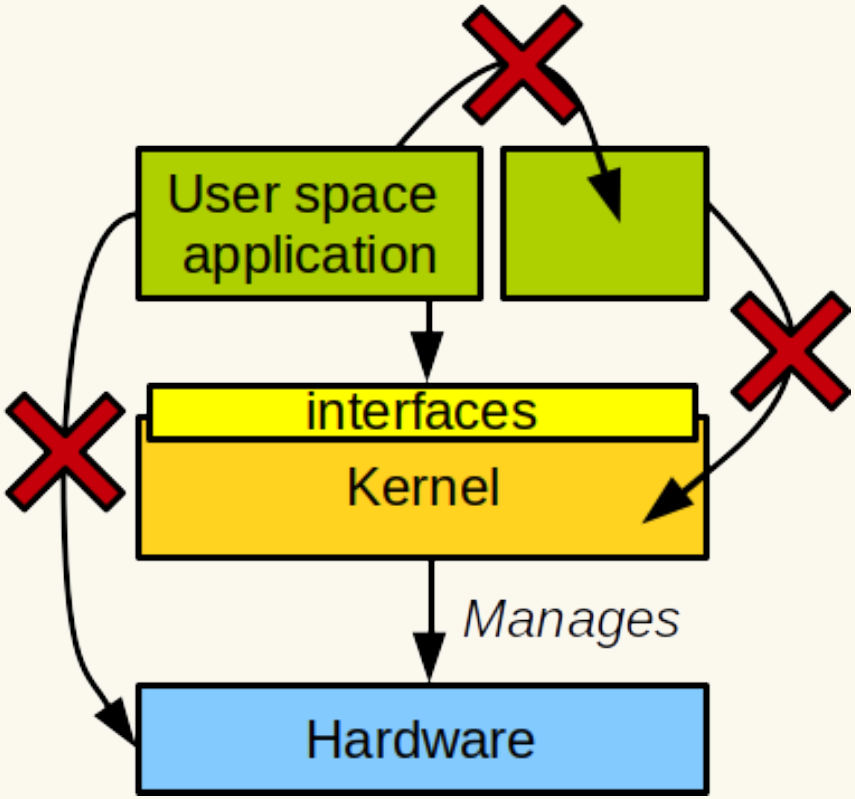
3 Syscall execution: kernel space side

4 Implementing a new system call

Virginia Tech

System calls: general notions

Kernel entry point from user space



- The kernel:
  - Manages the hardware
  - Provides **interfaces** or user space processes to access the hardware and perform privileged operations
- **User space cannot access HW/perform privilege operations directly**

Interfaces + user space privileges restriction: **the key to stability and security in the system**

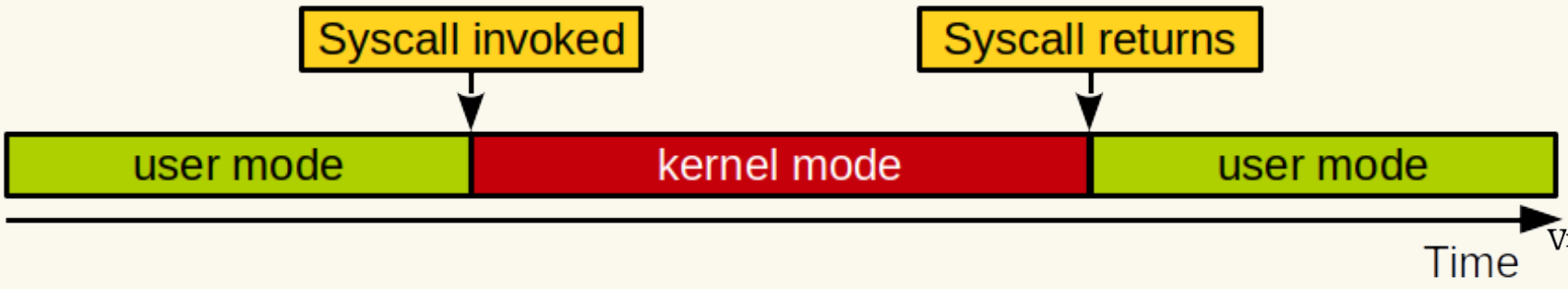
Virginia Tech

System calls: general notions

User/kernel mode

System calls (*syscalls*) are the one and only way an application can enter the kernel to request OS services and privileged operations such as accessing the hardware

- Examples of privileged/restricted operations:
  - **Privileged CPU instructions** (x86 examples): `HLT`, `INVLPLG`, `MOV` to control registers, etc.
    - Including IO related instructions (`IN`/`OUT`)
  - **Access to all memory areas**
    - Including areas mapping device registers



System calls: general notions

Examples of syscalls

Syscalls can be classified into groups:

- **Process management/scheduling:** `fork`, `exit`, `execve`, `nice`, `{get|set}priority`, `{get|set}pid`, etc.
- **Memory management:** `brk`, `mmap`, `swap{on|off}`, etc.
- **File system**: `open`, `read`, `write`, `lseek`, `stat`, etc.
- **Inter-Process Communication**: `pipe`, `shmget`, `semget`, etc.
- **Time management**: `{get|set}timeofday`, `time`, `timer_create`, etc.
- Others: `{get|set}uid`, `syslog`, `connect`, etc.

Virginia Tech

System calls: general notions

System calls table syscall identifier

- For x86_64, the syscall list is present in `arch/x86/syscalls/syscall_64.tbl` (4.0, location changes with versions)
  - Text file translated to c source code by a script during the compilation process

```
1  ## 64-bit system call numbers and entry vectors
2  #
3  # The format is:
4  # <number> <abi> <name> <entry point>
5  #
6  # The abi is "common", "64" or "x32" for this file.
7  #
8  0 common  read      sys_read
9  1 common  write     sys_write
10 2 common  open      sys_open
11 3 common  close     sys_close
12 # ...
```

  - Syscall identifier: unique integer
    - Currently 352 (linux 4.9) for x86_64
    - New syscalls identifiers are given sequentially

Virginia Tech

Outline

1 System calls: general notions

2 Syscall invocation: user space side

3 Syscall execution: kernel space side

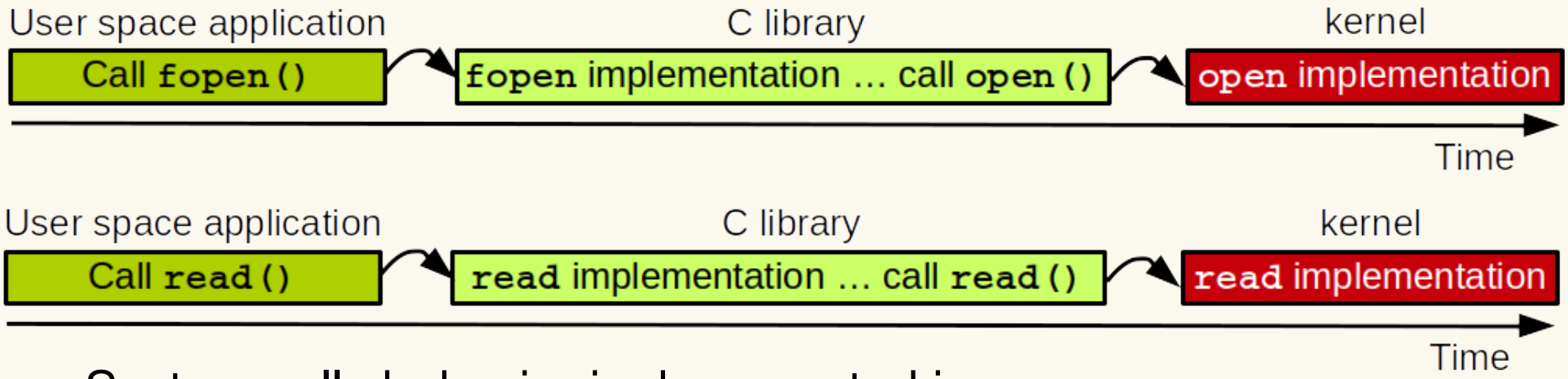
4 Implementing a new system call

Virginia Tech

Syscall invocation: user space side

C library

- Syscalls are rarely invoked directly
  - Most of them are wrapped by the C library
  - The programmer uses the C library **Application Programming Interface (*API*)**



- System calls behavior is documented in man pages

```
1 man <syscall name>
```

Syscall invocation: user space side

C library: Invocation without wrapper

- Some syscalls does not have a wrapper in the C library
- A syscall can be called directly through `syscall`
  - `man syscall`

```c
#include <unistd.h>
#include <sys/syscall.h>   /* For SYS_xxx definitions */

int main(void)
{
  char message[] = "hello, world!\n";
  int bytes_written = -42;

  /* the first "1" is the "write" syscall identifier */
  /* the second "1" is the standard output file descriptor */
  /* the remaining arguments are the "write" syscall arguments */
  bytes_written = syscall(1, 1, message, 14);

  /* or */

  bytes_written = syscall(SYS_write, 1, message, 14);

  return 0;
}
```

syscall.c

Syscall invocation: user space side

Invocation without the C library

- On x86_64, syscalls can be used directly through the `syscall` assembly instruction
  - Usage example: disabling the C library considerably reduces the size of a program

```
1         .global _start
2
3         .text
4 _start:
5         # write(1, message, 14)
6         mov     $1, %rax
7         mov     $1, %rdi
8         mov     $message, %rsi
9         mov     $14, %rdx
10        syscall
11
12        # exit(0)
13        mov     $60, %rax
14        xor     %rdi, %rdi
15        syscall
16 message:
17        .ascii  "Hello, world!\n"
```

syscall_asm.s

- Compilation & execution:

```
1 gcc -c syscall_asm.s
2     -o syscall_asm.o
3 ld syscall_asm.o
4     -o syscall_asm
5 ./syscall_asm
6 hello, world!
```

- Parameters are passed in registers

Outline

1 System calls: general notions

2 Syscall invocation: user space side

3 Syscall execution: kernel space side

4 Implementing a new system call

Virginia Tech

Syscall execution: kernel space side

User/kernel space transition

- **User space applications cannot call kernel code directly**
  - For security and stability, kernel code resides in a memory space that cannot be accessed from user space
  - *So how is a syscall invoked from user space ?*

Syscall execution: kernel space side

User/kernel space transition (2)

- A few words about interrupts:
  1. Asynchronous: **hardware interrupts**, issued from devices
     - Ex: keyboard indicating that a key has been pressed
  2. Synchronous: **exceptions**, triggered involuntarily by the program itself
     - Ex: divide by zero, page fault, etc.
  3. Synchronous, programmed exceptions: **software interrupts**, issued voluntarily by the code of the program itself
     - `INT` instruction for x86
- When an interrupt is received by the CPU, it stops whatever it is doing and **the kernel executes the interrupt handler**

Virginia Tech

Syscall execution: kernel space side

User/kernel space transition (3)

- ***So how is a syscall invoked from user space ?***
  - User space put the **syscall identifier and parameters values into registers** (x86)
    - Identifier in `rax`
    - x86_64: parameters in `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9`
  - Then issues a **software interrupt**
    - On x86, interrupt 128 was used:

```
1 int $0x80
```

    - Now `sysenter` (x86_32) and `syscall` (x86_64)
  - The kernel executes the interrupt handler, **system call handler**
    - Puts the registers values into a data structure placed on the stack
    - Checks the validity of the syscall (number of arguments)
    - Then execute the system call implementation:

```
1 call sys_call_table(, %rax, 8)
```

Virginia Tech

Syscall execution: kernel space side

Syscall implementation execution: example with gettimeofday

- Example: **gettimeofday**
  - implementation in sys_gettimeofday

```
NAME
       gettimeofday, settimeofday - get / set time

SYNOPSIS
       #include <sys/time.h>

       int gettimeofday(struct timeval *tv, struct timezone *tz);

       int settimeofday(const struct timeval *tv, const struct timezone *tz);

DESCRIPTION
       The functions gettimeofday() and settimeofday() can get and set the time as well as a
     timezone.  The tv argument is a struct timeval (as specified in <sys/time.h>):

           struct timeval {
               time_t      tv_sec;     /* seconds */
               suseconds_t tv_usec;    /* microseconds */
           };

       and gives the number of seconds and microseconds since the Epoch.
```

Syscall execution: kernel space side

Syscall implementation execution: example with gettimeofday (2)

- Usage example:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(void)
{
  struct timeval tv;
  int ret;

  ret = gettimeofday(&tv, NULL);
  if(ret == -1)
  {
    perror("gettimeofday");
    return EXIT_FAILURE;
  }

  printf("Local time:\n");
  printf(" sec:%lu\n", tv.tv_sec);
  printf(" usec:%lu\n", tv.tv_usec);

  return EXIT_SUCCESS;
}
```

```
./gettimeofday
Local time:
 sec:1485214886
 usec:523511
```

Syscall execution: kernel space side

Syscall implementation execution: example with gettimeofday (3)

- Let's check it out using vim code indexing features

```
1  SYSCALL_DEFINE2(gettimeofday, struct timeval __user
       *, tv,  struct timezone __user *, tz)
2  {
3    if (likely(tv != NULL)) {
4      struct timeval ktv;
5      do_gettimeofday(&ktv);
6      if (copy_to_user(tv, &ktv, sizeof(ktv)))
7        return -EFAULT;
8    }
9    if (unlikely(tz != NULL)) {
10     if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
11       return -EFAULT;
12   }
13   return 0;
14 }
```

- SYSCALL_DEFINE2
  - Macro to define sys_gettimeofday (2 parameters)
- **likely/unlikely**
  - Compiler assisted branch predictor hints

- **__user pointers and copy_{to|from}_user**
  - Kernel / user space memory management

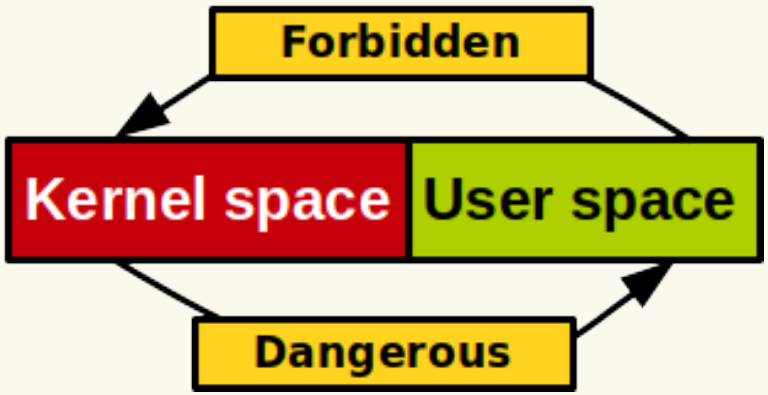
Syscall execution: kernel space side

likely/unlikely and kernel/user memory transfers

- **likely/unlikely**
  - include/linux/compiler.h:

```
1 #define likely(x)   (__builtin_expect(!!(x), 1)) /* !! convert to int and */
2 #define unlikely(x) (__builtin_expect(!!(x), 0)) /* into actual 0 or 1    */
```

- **User vs kernel memory areas**



- User space cannot access kernel memory
- Kernel code should not directly access user memory
- How to exchange data with pointers ?

Virginia Tech

Syscall execution: kernel space side

likely/unlikely and kernel/user memory transfers (2)

- The __user attribute
  - include/linux/compiler.h:

```
1 #define __user      __attribute__((noderef, address_space(1)))
2 #define __kernel    __attribute__((address_space(0)))
```

  - Used for static code security analysis (sparse)
- copy_{to|from}_user

```
1 static inline long copy_from_user(void *to, const void __user *from, unsigned long n);
2 static inline long copy_to_user(void __user *to, const void *from, unsigned long n);
```

  - When a kernel function gets a pointer to some memory in user space it needs to use:
    - The kernel copies it into its memory area (copy_from_user)
  - When the kernel wants to write in a user space buffer:
    - It uses copy_to_user
- These functions perform checks for security and stability

Outline

1 System calls: general notions

2 Syscall invocation: user space side

3 Syscall execution: kernel space side

4 Implementing a new system call

Implementing a new system call

Basic steps

1. **Write your syscall function**
   1. In an existing file if it makes sense
      - Is it related to time management ? → `kernel/time/time.c`
   2. Or, if the implementation is large and self-contained: in a new file
      - You will have to edit the kernel `Makefiles` to integrate it in the compilation process
2. **Add it to the syscall table and give it an identifier**
   - `arch/x86/syscalls/syscall_64.tbl` for Linux 4.0
3. **Add the prototype in `include/linux/syscalls.h`:**

```
1 asmlinkage long sys_gettimeofday(struct timeval __user *tv,
2                                  struct timezone __user *tz);
```

4. Recompile, reboot and run
   - Touching the syscall table will trigger the entire kernel compilation

Virginia Tech

Implementing a new system call

Editing the kernel Makefiles (2)

- Example: syscall implemented in linux sources in `my_syscall/my_syscall.c`
  1. `my_syscall/Makefile`:

```
1 obj-y += my_syscall.o
```

  2. Linux root `Makefile`:

```
1 # ...
2 core-y          += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ my_syscall/
3 # ...
```

Implementing a new system call

Do you really need it?

- **Pros:** Easy to implement and use, fast
- **Cons:**
  - Needs an official syscall number
  - Interface cannot change after implementation
  - Must be registered for each architecture
  - Probably too much work for small exchanges of information
- **Alternative:**
  - Device or virtual file:
    - User/kernel space communication through `read`, `write`, `ioctl`

Virginia Tech